Chapter 9

Optimization Spaces and Dimensions for Self-evolution

The optimization of autonomous agents represents a complex challenge that encompasses multiple levels of abstraction. In this section, we first establish prompt optimization as the foundational layer, upon which three distinct branches of optimization emerge: agentic workflow optimization, tool optimization, and comprehensively autonomous agent optimization.

9.1 Overview of Agent Optimization

Existing LLM-based agent optimization can be conceptualized in terms of a two-tiered architecture. At the foundation lies *prompt optimization*, which focuses on enhancing the basic interaction patterns of Language Model nodes. Building upon this foundation, three parallel branches emerge: i) *workflow-level optimization*, which focuses on the coordination and interaction patterns between multiple LLM nodes; ii) *tool optimization*, where agents evolve by developing and improving tools to adapt to new tasks and leverage past data; and iii) *comprehensive autonomous agent optimization*, which aims at the holistic enhancement of agent capabilities by considering multiple dimensions.

Similarly to optimization paradigms in AutoML, agent optimization can be categorized as either single-objective or multi-objective. Contemporary agent optimization primarily centers on three canonical metrics: performance, inference cost, and latency. Performance measures the effectiveness of the agent in completing its assigned tasks, while inference cost quantifies the computational resources required for agent operation. Latency represents the time taken for the agent to respond and complete tasks. These objectives can vary depending on the specific optimization modality. For instance, in prompt-level optimization, additional constraints such as prompt length may become relevant objectives. This multi-faceted nature of optimization objectives reflects the complexity of agent systems and the need to balance multiple competing requirements.

9.2 Prompt Optimization

Prompt optimization plays the most critical role in LLM-based agent optimization. When optimizing an agent, beyond model-level optimizations, task-specific or model-specific prompt optimization directly impacts the agent's performance, latency, and cost. Given a task $T = (Q, G_t)$, where Q denotes the input query and G_t represents the optional ground truth, the objective of prompt optimization is to generate a task-specific prompt P_t^* that maximizes performance:

$$P^* = \arg\max_{P \in \mathcal{P}} \mathbb{E}_{T \sim \mathcal{D}}[\phi_{\text{eval}}(\phi_{\text{exe}}(Q, P), T)] \tag{9.1}$$

where $\mathcal{P}$ represents the space of possible prompts, ϕ_{exe} denotes the execution function, and ϕ_{eval} represents the evaluation function. This optimization is typically implemented through three fundamental functions: ϕ_{opt}, ϕ_{exe}, and ϕ_{eval}. The Optimize function ϕ_{opt} refines existing prompts based on optimization signals, the Execute function ϕ_{exe} invokes the current prompt to obtain output O, and the Evaluation function ϕ_{eval} assesses current outputs to generate evaluation signals S_{eval} and optimization signals S_{opt}. The evaluation signals are used to select effective prompts, while the optimization signals assist the Optimize function in performing optimization.

9.2.1 Evaluation Functions

At the core of prompt optimization lies the evaluation function ϕ_{eval}, which serves as the cornerstone for deriving optimization signals and guiding the evolutionary trajectory of prompts. This function orchestrates a sophisticated interplay between evaluation sources, methodologies, and signal generation, establishing a feedback loop that drives continuous improvement. The evaluation function ϕ_{eval} processes evaluation sources as input, and employs various evaluation methods to generate different types of signals, which subsequently guide the optimization process. Here, we define the dimensions of sources, methods, and signal types to establish the foundation for prompt optimization.

Evaluation Sources Evaluation sources primarily consist of LLM Generated Output G_{llm} and task-specific Ground Truth G_t. Existing works such as [730, 774, 728, 775, 732, 300] predominantly leverage comparisons between G_{llm} and G_t as evaluation sources. Some approaches [776, 721, 777] utilize only G_{llm} as the evaluation source. For instance, PROMST [721] assesses prompt effectiveness by comparing G_{llm} against human-crafted rules; SPO [778] employs pairwise comparisons of outputs from different prompts to determine relative effectiveness.

Evaluation Methods Evaluation Methods can be broadly categorized into three approaches: *benchmark-based evaluation*, *LLM-as-a-Judge*, and *human feedback*. *Benchmark-based evaluation* remains the most prevalent method in prompt optimization [730, 774, 721, 732, 300]. This approach relies on predefined metrics or rules to provide numerical feedback as evaluation signals. While it offers an automated evaluation process, its effectiveness ultimately depends on how well the benchmark design aligns with human preferences.

The introduction of *LLM-as-a-Judge* represents a significant advancement in automated evaluation and preference alignment. Leveraging LLMs' inherent alignment with human preferences and carefully designed judging criteria, this approach [589] can assess task completion quality based on task descriptions and prompt outputs G_{llm}, providing reflective textual gradient feedback. Notable implementations include ProteGi [779], TextGrad [728], Semantic Search [775] and Revolve [780]. Furthermore, LLM-as-a-judge enables comparative evaluation between ground truth G_t and output G_{llm} with specific scoring mechanisms [724]. The effectiveness of this method hinges on both the design of judger prompts and the underlying model's alignment with human preferences. As a specialized extension, *Agent-as-a-Judge* [781] refines this paradigm by employing dedicated agents for providing process evaluation on complex tasks, while maintaining high alignment with human preferences at significantly reduced evaluation costs.

Human feedback represents the highest level of intelligence integration in the evaluation process. As humans remain the ultimate arbiters of prompt effectiveness, direct human feedback can rapidly and substantially improve prompt quality. However, this approach introduces significant resource overhead. APOHF [777] demonstrates that incorporating human feedback can achieve robust prompt optimization with minimal computational resources, particularly excelling in open-ended tasks such as user instructions, prompt optimization for text-to-image generative models, and creative writing. Nevertheless, the requirement for human intervention somewhat contradicts the goal of automated evolution.

Signal Types Feedback generated by evaluation methods manifests in three distinct forms, each serving different optimization needs. *Numerical feedback* [730, 774, 721, 732, 300] quantifies performance through scalar metrics, compatible with rules, ground truth, human assessment, and LLM judgments. While widely applicable, this approach requires substantial samples for statistical reliability, potentially overlooking instance-specific details that could guide optimization. *Textual feedback* [728, 775, 780] provides detailed, instance-specific guidance through analysis and concrete suggestions. This sophisticated approach requires intelligent participation, either from human experts or advanced language models, enabling targeted improvements in prompt design through explicit recommendations. However, its reliance on sophisticated intelligence sources impacts its scalability.*Ranking feedback* [778] establishes relative quality ordering through either comprehensive ranking or pairwise comparisons. This approach uniquely circumvents the need for absolute quality measures or predefined criteria, requiring only preference judgments. It proves particularly valuable when absolute metrics are difficult to define or when optimization primarily concerns relative improvements.

9.2.2 Optimization Functions

The design of optimization functions is crucial in determining the quality of generated prompts in each iteration of prompt optimization. Through effective signal guidance, prompt self-evolution can achieve faster convergence. Current optimization approaches primarily rely on two types of signals: *evaluation signals* S_{eval} that identify the most effective existing prompts, and *optimization signals* S_{opt} that provide detailed guidance for improvements.

Optimize via Evaluation Signals When optimizing with evaluation signals, the process begins by selecting the most effective prompts based on ϕ_{eval} assessments. Rather than directly learning from past errors, some methods adopt heuristic exploration and optimization strategies. SPO [778] iteratively refines prompts based on the outputs of current best-performing ones, leveraging the language model's inherent ability to align with task requirements. Similarly, Evoprompt [723] employs evolutionary algorithms with LLMs serving as evolution operators for heuristic prompt combination. PromptBreeder [732] advances this approach further by comparing score variations between mutated prompts while simultaneously modifying both meta-prompts and prompts through the LLM's inherent capabilities.

Optimize via Optimization Signals While optimization methods based solely on evaluation signals require extensive search to find optimal solutions in vast search spaces through trial and error, an alternative approach leverages explicit optimization signals to guide the optimization direction and improve efficiency. Existing methods demonstrate various ways to utilize these optimization signals. OPRO [730] extracts common patterns from high-performing prompt solutions to guide subsequent optimization steps. ProTegi [779] employs language models to analyze failure cases and predict error causes, using these insights as optimization guidance. TextGrad [728] extends this approach further by transforming prompt reflections into "textual gradients", applying this guidance across multiple prompts within agentic systems. Revolve [780] further enhances optimization by simulating second-order optimization, extending previous first-order feedback mechanisms to model the evolving relationship between consecutive prompts and responses. This allows the system to adjust based on how previous gradients change, avoiding stagnation in suboptimal patterns and enabling more informed, long-term improvements in complex task performance.

9.2.3 Evaluation Metrics

The effectiveness of prompt optimization methods can be evaluated across multiple dimensions. *Performance metrics* [782, 778, 730] for Close Tasks serve as the most direct indicators of a prompt's inherent performance, encompassing measures such as pass@1, accuracy, F1 score, and ROUGE-L. These metrics enable researchers to assess the stability, effectiveness, and convergence rate of prompt optimization processes. Another crucial dimension is *Efficiency metrics* [778]. While some prompt optimization approaches achieve outstanding results, they often demand substantial computational resources, larger sample sizes, and extensive datasets. In contrast, other methods achieve moderate results with lower resource requirements, highlighting the trade-offs between performance and efficiency in agent evolution. The third dimension focuses on qualitative metrics that assess specific aspects of agent behavior: consistency [776] measures output stability across multiple runs, fairness [783] evaluates the ability to mitigate the language model's inherent biases, and confidence [784, 785] quantifies the agent's certainty in its predictions. When these behavioral aspects are treated as distinct objectives, prompt optimization frameworks provide corresponding metrics for evaluation.

9.3 Workflow Optimization

While prompt-level optimization has shown promising results in enhancing individual LLM capabilities, modern AI systems often require the coordination of multiple LLM components to tackle complex tasks. This necessitates a more comprehensive optimization domain—the agentic workflow space. At its core, an agentic workflow consists of LLM-invoking nodes, where each node represents a specialized LLM component designed for specific sub-tasks within the larger system.

Although this architecture bears similarities to multi-agent systems, it is important to distinguish agentic workflows from fully autonomous multi-agent scenarios. In agentic workflows, nodes operate under predetermined protocols and optimization objectives, rather than exhibiting autonomous decision-making capabilities. Many prominent systems, such as MetaGPT [626] AlphaCodium [786] can be categorized under this framework. Moreover, agentic workflows can serve as executable components within larger autonomous agent systems, making their optimization crucial for advancing both specialized task completion and general agent capabilities.

Following the formalization proposed by GPTSwarm [651] and AFLOW [773], this section first establishes a formal definition of agentic workflows and their optimization objectives. We then examine the core components of agentic workflows—nodes and edges—analyzing their respective search spaces and discussing existing representation approaches in the literature.

9.3.1 Workflow Formulation

An agentic workflow $\mathcal{K}$ can be formally represented as:

$$\mathcal{K} = \{(N, E) | N \in \mathcal{N}, E \in \mathcal{E}\} \tag{9.2}$$

where $\mathcal{N} = \{N(M, \tau, P, F) | M \in \mathcal{M}, \tau \in [0, 1], P \in \mathcal{P}, F \in \mathcal{F}\}$ represents the set of LLM-invoking nodes, with M, τ, $\mathcal{P}$, and $\mathcal{F}$ denoting the available language models, temperature parameter, prompt space, and output format space respectively. E indicates the edges between different LLM-invoking nodes. This formulation encapsulates both the structural components and operational parameters that define an agentic workflow's behavior.

Given a task T and evaluation metrics L, the goal of workflow optimization is to discover the optimal workflow K^* that maximizes performance:

$$K^* = \arg\max_{K \in \mathcal{K}} L(K, T) \tag{9.3}$$

where K is the search space of workflow, and $L(K, T)$ typically measures multiple aspects including task completion quality, computational efficiency, and execution latency. This optimization objective reflects the practical challenges in deploying agentic workflows, where we must balance effectiveness with resource constraints.

9.3.2 Optimizing Workflow Edges

The edge space $\mathcal{E}$ defines the representation formalism for agentic workflows. Current approaches primarily adopt three distinct representation paradigms: graph-based, neural network-based, and code-based structures. Each paradigm offers unique advantages and introduces specific constraints on the optimization process.

Graph-based representations enable the expression of hierarchical, sequential, and parallel relationships between nodes. This approach naturally accommodates complex branching patterns and facilitates visualization of workflow topology, making it particularly suitable for scenarios requiring explicit structural manipulation. For example, GPTSwarm [651] demonstrated the effectiveness of graph-based workflow representation in coordinating multiple LLM components through topology-aware optimization. Neural network architectures provide another powerful representation paradigm that excels in capturing non-linear relationships between nodes. Dylan [725] showed that neural network-based workflows can exhibit adaptive behavior through learnable parameters, making them especially effective for scenarios requiring dynamic adjustment based on input and feedback. Code-based representation offers the most comprehensive expressiveness among current approaches. AFLOW [773] and ADAS [741] established that representing workflows as executable code supports linear sequences, conditional logic, loops, and the integration of both graph and network structures. This approach provides precise control over workflow execution and leverages LLMs' inherent code generation capabilities.

The choice of edge space representation significantly influences both the search space dimensionality and the applicable optimization algorithms. [728] focused solely on prompt optimization while maintaining a fixed workflow topology, enabling the use of textual feedback-based optimization techniques. In contrast, [651] developed reinforcement learning algorithms for joint optimization of individual node prompts and overall topology. [773] leveraged code-based representation to enable direct workflow optimization by language models, while recent advances by [787] and [788] introduced methods for problem-specific topology optimization.

9.3.3 Optimizing Workflow Nodes

The node space N consists of four key dimensions that influence node behavior and performance. The output format space F significantly impacts performance by structuring LLM outputs, with formats like XML and JSON enabling more precise control over response structure. The temperature parameter τ controls output randomness, affecting the stability-creativity tradeoff in node responses. The prompt space P inherits the optimization domain from prompt-level optimization, determining the core interaction patterns with LLMs. The model space M represents available LLMs, each with distinct capabilities and computational costs.

For single-node optimization, existing research has primarily focused on specific dimensions within this space. [773] concentrated exclusively on prompt optimization, while [741] extended the search space to include both prompts and temperature parameters. Taking a different approach, [789] fixed prompts while exploring model selection across different nodes. Output format optimization, though crucial, remains relatively unexplored [790].

Compared to edge space optimization, node space optimization poses unique scalability challenges due to the typically large number of nodes in agentic workflows. The dimensionality of the search space grows multiplicatively with each additional node, necessitating efficient optimization strategies that can effectively handle this complexity while maintaining reasonable computational costs.

9.4 Tool Optimization

Unlike conventional usage of LLMs that typically operate in a single-turn manner, agents are equipped with advanced multi-turn planning capabilities and the ability to interact with the external world via various tools. These unique attributes make the optimization of tool usage a critical component in enhancing an agent's overall performance and adaptability. Tool optimization involves systematically evaluating and refining how an agent selects, invokes, and integrates available tools to solve problems with higher efficiency and lower latency. Key performance metrics in this context include decision-making accuracy, retrieval efficiency, selection precision, task planning, and risk management. Central to this optimization are two complementary strategies: *tool learning* and *tool creation*.

9.4.1 Learning to Use Tools

Unlike prompting-based methods that leverage frozen foundation models' in-context learning abilities, training-based methods optimize the model that backs LLM agents with supervision. Drawing inspiration from developmental psychology, tool learning can be categorized into two primary streams: *learning from demonstrations* and *learning from feedback* [714]. The other way to elicit the power of LLMs (agents) using tools is by using prompt-based or in-context learning methods for better reasoning abilities.

Learning from demonstrations involves training models backed LLM agents to mimic expert behaviors through imitation learning. Techniques such as behavior cloning allow models to learn policies in a supervised manner by replicating human-annotated tool-use actions. Formally, given a dataset $D = \{(q_i, a_i^*)\}_{i=0}^{N-1}$, where q_i is a user query and a_i^* is the corresponding human demonstration, the controller's parameters θ_C are optimized as:

$$\theta_C^* = \arg\max_{\theta_C} \mathbb{E}_{(q_i,a_i^*)\in D} \prod_{t=0}^{T_i} p_{\theta_C}(a_{i,t}^* \mid x_{i,t}, H_{i,t}, q_i)$$

where $a_{i,t}^*$ is the human annotation at timestep t for query q_i, and T_i is the total number of timesteps.

Learning from feedback leverages reinforcement learning to enable models to adapt based on rewards derived from environment or human feedback. The optimization objective for the controller's parameters θ_C is:

$$\theta_C^* = \arg\max_{\theta_C} \mathbb{E}_{q_i\in Q} \mathbb{E}_{\{a_{i,t}\}_{t=0}^{T_i}} \left[R\left(\{a_{i,t}\}_{t=0}^{T_i} \right) \right]$$

where R represents the reward function based on the sequence of actions $\{a_{i,t}\}$.

Integrating tool learning into the optimization framework enhances the system's ability to generalize tool usage across diverse tasks and environments. By incorporating both demonstration-based and feedback-based learning, the model can iteratively improve its tool invocation strategies, selection policies, and execution accuracy.

Optimization Reasoning Strategies for Tool Using Optimizing the aforementioned metrics for better LLM agents' abilities requires a combination of advanced retrieval models, fine-tuned reasoning strategies, and adaptive learning mechanisms. Reasoning strategies, such as Chain-of-Thought (CoT) [46], Tree-of-Thought [72], and Depth-First Search Decision Trees (DFS-DT) [690], facilitate more sophisticated decision-making processes regarding tool usage. Fine-tuning the model's understanding of tools, including parameter interpretation and action execution, enables more precise and effective tool interactions. Additionally, learning from the model's outputs allows for better post-processing and analysis, further refining tool utilization efficacy.

9.4.2 Creation of New Tools

Beyond the optimization of existing tools, the ability to create new tools dynamically [703, 702, 772] based on a deep understanding of tasks and current tool usage can significantly enhance the LLM Agent framework's adaptability and efficiency. In recent work, several complementary approaches have been proposed. ToolMakers [702] establishes a closed-loop framework where a tool-making agent iteratively executes three phases: (1) *Proposing* Python functions via programming-by-example using three demonstrations, (2) *Verifying* functionality through automated unit testing (3 validation samples) with self-debugging of test cases, and (3) *Wrapping* validated tools with usage demonstrations for downstream tasks. This rigorous process ensures reliability while maintaining full automation. CREATOR [703] adopts a four-stage lifecycle: *Creation* of task-specific tools through abstract reasoning, *Decision* planning for tool invocation, *Execution* of generated programs, and *Rectification* through iterative tool refinement—emphasizing tool diversity, separation of abstract/concrete reasoning, and error recovery mechanisms. In contrast, CRAFT [772] employs an offline paradigm that distills domain-specific data into reusable, atomic tools (e.g., object color detection) through GPT-4 prompting, validation, and deduplication. Its training-free approach combines human-inspectable code snippets with compositional problem-solving, enabling explainable toolchains while avoiding model fine-tuning—particularly effective when decomposing complex tasks into modular steps.

The integration of these complementary approaches presents rich research opportunities. Hybrid systems could merge CRAFT's pre-made tool repositories with ToolMakers' on-demand generation, using functional caching to balance efficiency and adaptability. Future frameworks might implement multi-tier tool hierarchies where primitive operations from CRAFT feed into ToolMakers' composite tools, while CREATOR-style rectification handles edge cases. Advances in self-supervised tool evaluation metrics and cross-domain generalization could further automate the tool lifecycle. Notably, the interplay between tool granularity (atomic vs. composite) and reusability patterns warrants systematic investigation—fine-grained tools enable flexible composition but increase orchestration complexity. As agents evolve, bidirectional tool-task co-adaptation mechanisms may emerge, where tools reshape task representations while novel tasks drive tool innovation, ultimately enabling self-improving AI systems.

9.4.3 Evaluation of Tool Effectiveness

The evaluation metrics and benchmarks discussed below offer a comprehensive basis for quantifying an agent's tool usage capabilities. By assessing aspects such as tool invocation, selection accuracy, retrieval efficiency, and planning for complex tasks, these benchmarks not only measure current performance but also provide clear, concrete objectives for optimizing tool usage. Such metrics are instrumental in guiding both immediate performance enhancements and long-term strategic improvements in agent-based systems. In the following sections, we first review the evolution of agent tool use benchmarks and then consolidate the key evaluation metrics that serve as targets for further tool optimization.

Tool Evaluation Benchmarks Recent efforts in LLM-as-Agent research have spawned diverse benchmarks and frameworks for evaluating tool-use capabilities. Early studies such as Gorilla [727] and API-Bank [791] pioneered large-scale datasets and methods for testing LLM interactions with external APIs, shedding light on issues like argument accuracy and hallucination. Subsequent works like T-Bench [792] and ToolBench [690] introduced more extensive task suites and stressed the importance of systematic data generation for tool manipulation. StableToolBench [793] further extended this line of inquiry by highlighting the instability of real-world APIs, proposing a virtual API server for more consistent evaluation. Meanwhile, ToolAlpaca [794] investigated the feasibility of achieving generalized tool-use in relatively smaller language models with minimal in-domain training. Additional efforts like ToolEmu [795] assessed the safety and risk aspects of tool-augmented LM agents through emulated sandbox environments. MetaTool [796] then introduced a new benchmark focused on whether LLMs know *when* to use tools and can correctly *choose* which tools to employ. It provides a dataset named ToolE that covers single-tool and multi-tool usage scenarios, encouraging research into tool usage awareness and nuanced tool selection. ToolEyes [797] pushed the evaluation further by examining real-world scenarios and multi-step reasoning across a large tool library. Finally, τ-bench [798] introduced a human-in-the-loop perspective, emphasizing dynamic user interactions and policy compliance in agent-based conversations. Together, these benchmarks and frameworks underscore the evolving landscape of tool-augmented LLM research, marking a shift from isolated reasoning tasks to comprehensive, real-world agent evaluations.

Metrics for Tool Invocation Deciding whether to invoke an external tool is a critical step that can significantly affect both the efficiency and the effectiveness of a system. In many scenarios, the model must determine if its own reasoning is sufficient to answer a query or if additional external knowledge (or functionality) provided by a tool is required. To formalize this process, we introduce a labeled dataset

$$D_{\text{inv}} = \{(q_i, y_i)\}_{i=0}^{N-1},$$

where q_i represents the i-th user query and $y_i \in \{0, 1\}$ is a binary label indicating whether tool invocation is necessary ($y_i = 1$) or not ($y_i = 0$). Based on this dataset, the model learns a decision function $d(q_i)$ defined as:

$$d(q_i) = \begin{cases} 1, & \text{if } P_\theta(y = 1 \mid q_i) \geq \tau, \\ 0, & \text{otherwise,} \end{cases}$$

where $P_\theta(y = 1 \mid q_i)$ denotes the predicted probability (from a model parameterized by θ) that a tool should be invoked for query q_i, and τ is a predetermined threshold.

In addition to this decision rule, several metrics can be used to evaluate the model's ability to correctly decide on tool invocation. For example, the overall invocation accuracy A_{inv} can be computed as:

$$A_{\text{inv}} = \frac{1}{N} \sum_{i=0}^{N-1} \mathbf{1}\{d(q_i) = y_i\},$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. Other metrics such as precision, recall, and F1 score are also applicable. Moreover, if C_{inv} represents the cost incurred by invoking a tool and $R(q_i)$ the benefit or reward obtained when a tool is correctly used, one can define a net benefit score:

$$B_{\text{inv}} = \sum_{i=0}^{N-1} \left(\mathbf{1}\{d(q_i) = 1\} \cdot R(q_i) - C_{\text{inv}} \right).$$

This formulation not only emphasizes accuracy but also considers the cost-effectiveness of invoking external tools.

Tool Selection Among Candidates Once the decision to invoke a tool is made, the next challenge is to select the most appropriate tool from a pool of candidates. Let the candidate toolset be represented as:

$$\mathcal{T} = \{t_1, t_2, \ldots, t_M\}.$$

For a given query q_i, assume that the optimal tool (according to ground truth) is t_i^* and the model selects $\hat{t}_i$. The simplest measure of selection performance is the tool selection accuracy A_S:

$$A_S = \frac{1}{|Q|} \sum_{q_i \in Q} \mathbf{1}\{\hat{t}_i = t_i^*\}.$$

However, many scenarios involve ranking multiple candidate tools by their relevance. In such cases, ranking-based metrics such as Mean Reciprocal Rank (MRR) and normalized Discounted Cumulative Gain (nDCG) offer a more nuanced evaluation. [690] use those two when evaluating the tool retriever system.

Tool Retrieval Efficiency and Hierarchical Accuracy Tool retrieval involves both the speed of identifying a suitable tool and the accuracy of that selection. Efficient retrieval methods reduce latency and computational overhead, while high retrieval accuracy ensures that the most relevant tool is identified for the task. To evaluate tool usage comprehensively, we adopt a hierarchical framework that distinguishes between retrieval accuracy and selection accuracy. Retrieval accuracy (A_R) reflects how precisely the system retrieves the correct tool from the repository, typically measured by metrics such as Exact Match (EM) and F1 score, which capture both complete and partial matches. In contrast, selection accuracy (A_S) assesses the system's ability to choose the optimal tool from a set of candidates, again using similar metrics. Overall tool usage awareness is further evaluated by accuracy, recall, precision, and F1 score.

The overall retrieval efficiency E_{Ret} is thus can be expressed as:

$$E_{Ret} = \frac{A_R \times A_S \times A_P \times A_U}{C_R}$$

where C_R is the cost associated with retrieval. Optimization strategies may involve training embedding models with feedback mechanisms to enhance both efficiency and each hierarchical component of accuracy.

For a more nuanced evaluation of tool selection, Metatool [796] introduces the Correct Selection Rate (CSR), which quantifies the percentage of queries for which the model selects the expected tool(s). This evaluation framework addresses four aspects: selecting the correct tool among similar candidates, choosing appropriate tools in context-specific scenarios, ensuring reliability by avoiding the selection of incorrect or non-existent tools, and handling multi-tool queries. Together, these metrics and sub-tasks provide a robust measure of both the efficiency and precision in tool retrieval and selection.

Tool Planning for Complex Tasks Complex tasks often require the sequential application of multiple tools to reach an optimal solution. A tool plan can be represented as an ordered sequence

$$\Pi = [t_1, t_2, \ldots, t_K],$$

where K is the number of steps. The quality of such a plan is typically evaluated by balancing its task effectiveness (e.g., via a metric $R_{\text{task}}(\Pi)$) against the plan's complexity (or length). This balance can be captured by a composite planning score of the form

$$S_{\text{plan}} = \alpha \cdot R_{\text{task}}(\Pi) - \beta \cdot K,$$

where α and β are coefficients that adjust the trade-off between the benefits of high task performance and the cost associated with plan complexity. When ground truth plans Π^* are available, similarity metrics such as BLEU or ROUGE can be used to compare the predicted plan Π with Π^*, and an overall planning efficiency metric can be defined accordingly.

In addition, recent work such as ToolEyes [797] highlights the importance of behavioral planning in tool usage. Beyond selecting tools and parameters, it is crucial for LLMs to concisely summarize acquired information and strategically plan subsequent steps. In this context, the behavioral planning capability is evaluated along two dimensions. First, the score $S_{b\text{-validity}} \in [0,1]$ is computed by assessing (1) the reasonableness of summarizing the current state, (2) the timeliness of planning for the next sequence of actions, and (3) the diversity of planning. Second, the score $S_{b\text{-integrity}} \in [0,1]$ is calculated by evaluating (1) grammatical soundness, (2) logical consistency, and (3) the ability to correct thinking. The composite behavioral planning score is then determined as

$$S_{BP} = S_{b\text{-validity}} \cdot S_{b\text{-integrity}},$$

providing a holistic measure of the model's planning capability. This integrated framework ensures that tool planning for complex tasks not only focuses on the selection and ordering of tools but also on maintaining coherent, effective, and strategically sound planning processes.

In summary, optimizing tool performance within an Agent system necessitates a comprehensive approach that balances decision-making accuracy, retrieval efficiency, hierarchical selection precision, strategic planning, rigorous risk management, and robust tool learning mechanisms. By implementing targeted optimization and learning strategies, it is possible to enhance both the effectiveness and efficiency of tool-assisted machine learning workflows.

9.5 Towards Autonomous Agent Optimization

In addition to optimizing individual modules in agent evolution, such as prompts, tools, and workflows—which are susceptible to local optima that can compromise the overall performance of the agentic system, a significant body of research focuses on optimizing multiple components within the entire agentic systems. This holistic approach enables large language model (LLM) agents to evolve more comprehensively. However, optimizing the entire system imposes higher requirements. The algorithm must not only account for the impact of individual components on the agentic system but also consider the complex interactions between different components.

ADAS [741] is one of the most representative works that first formally defines the research problem of automated design in agentic systems. It integrates multiple agentic system components into the evolutionary pipeline. Specifically, ADAS introduces a meta-agent capable of iteratively designing the agentic system's workflow, prompts, and potential tools within the overall optimization process. As demonstrated in the experiments, the automatically designed agentic systems outperform state-of-the-art hand-designed baselines.

Additionally, [726] proposes an agent symbolic learning framework for training language agents, inspired by connectionist learning principles used in neural networks. By drawing an analogy between agent pipelines and computational graphs, the framework introduces a language-based approach to backpropagation and weight updates. It defines a prompt-based loss function, propagates language loss through agent trajectories, and updates symbolic components accordingly. This method enables structured optimization of agentic workflows and naturally extends to multi-agent systems by treating nodes as independent agents or allowing multiple agents to act within a single node.

[799] proposes an approach to optimize both prompts and the agent's own code, enabling self-improvement. This aligns with the concept of self-reference, where a system can analyze and modify its own structure to enhance performance.

Similarly, [773], [787], [800] and [788] focus on optimizing both the workflow and prompts within agentic systems. In particular, [285] introduces an approach that trains additional large language models (LLMs) to generate prompts and workflows, enabling the automated design of agentic system architectures.

In summary, optimizing the workflow of an entire agentic system is not merely a straightforward aggregation of individual component optimizations. Instead, it requires carefully designed algorithms that account for complex interdependencies among components. This makes system-wide optimization a significantly more challenging task, necessitating advanced techniques to achieve effective and comprehensive improvements.